# Przetwarzanie rozproszone w SZBD Postgres

Celem zajęć jest zapoznanie się z własnościami SZBD Postgres umożliwiającymi przetwarzanie rozproszone. Własności te obejmują:

* Transparenty dostęp do zdalnych danych.
* Rozproszone transakcje.
* Obsługa rozproszonych zakleszczeń.
* Podstawowa replikacja danych.

## Przygotowanie środowiska

Do ilustracji przetwarzania rozproszonego będziemy korzystać ze środowiska Kubernetes (w skrócie K8s). Środowisko to służy do orkiestracji skonteneryzowanych, rozproszonych aplikacji o architekturze mikro-usługowej. Głównymi zadaniami orkiestracji są: wdrożenie skonteneryzowanych składników aplikacji, dynamiczne skalowanie aplikacji na żądanie, automatyczne odtwarzanie składników aplikacji, uaktualnianie i wycofywanie aktualizacji bezstanowych składników aplikacji z zachowaniem ciągłości ich pracy. Konteneryzacja aplikacji polega na uruchomieniu jej w kontenerach, które są lekką odmianą mechanizmu wirtualizacji umożliwiającą separację różnych składników aplikacji i kontrolę przydziału do nich zasobów (pamięć, procesor, dysk, sieć). Najpopularniejszym środowiskiem konteneryzacji jest Docker. Architektura mikro-usługowa jest sposobem do projektowania dużych systemów aplikacji, które wykorzystuje podejście podziału systemu na wiele składników, nazywanych mikro-usługami, komunikujących się ze sobą najczęściej asynchronicznie z wykorzystaniem systemów kolejkowych (np. Kafka, RabbitMQ), które w celu uzyskania skalowalności rozprasza się na wiele maszyn. W ramach zajęć skupimy się składnikach aplikacji, które służą do trwałego składowania rozproszonych danych, czyli na rozproszonych bazach danych.

Kubernetes jest otwarto-źródłowym systemem rozwijanym przez Google. Jego usługi są oferowane przez największych dostawców rozwiązań chmurowych, np.: Google Cloud, Amazon AWS, Microsoft Azure, Alibaba Cloud oraz przez mniejszych dostawców np. Hosted Rancher. Kubernetes można też wdrożyć na własnych serwerach. W celu uniezależnienia się od infrastruktury chmurowej i sprzętowej wykorzystamy jedną z dystrybucji deweloperskich Kubernetes o nazwie *k3d*, którą można uruchomić na sprzęcie klasy desktop. *K3d* w odróżnieniu od innych dystrybucji deweloperskich Kubernetes np. *k3s* lub *Minikube* symuluje wiele węzłów klastra co umożliwia zwiększenie realizmu. Pliki manifestów opisujące sposób wdrożenia aplikacji przygotowane dla *k3d* po nieznacznych uzupełnieniach można wykorzystać w realnym środowisku chmurowym. *K3d* ma architekturę matrioszki, jest uruchamiany jako zbiór kontenerów Docker, każdy węzeł klastra *k3d* posiada własny kontener. Wewnątrz tych kontenerów są uruchamiane kontenery zawierające *k3s*, z kolei w tych kontenerach są uruchamiane kontenery udostępniające funkcjonalność Kubernetes i kontenery użytkownika.

Przygotowanie środowiska rozpoczniemy od zainstalowania Docker i *k3d*.

1. Zaloguj się do maszyny wirtualnej jako użytkownik rbd używając hasła rbd#2502.
2. Otwórz okno terminala, który nazwiemy terminalem pomocniczym.
3. Skonfiguruj menedżera pakietów do zainstalowania pakietu Doker, w tym celu wykonaj poniższe polecenia.

sudo apt-get update

sudo apt-get -y install ca-certificates curl

sudo install -m 0755 -d /etc/apt/keyrings

sudo curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc

sudo chmod a+r /etc/apt/keyrings/docker.asc

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/debian \

$(. /etc/os-release && echo "$VERSION\_CODENAME") stable" | \

sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

1. Zainstaluj Docker wykonując w terminalu pomocniczym poniższe polecenia:  
   apt-get -y install docker-ce=5:28.4.0-1~debian.13~trixie \

docker-ce-cli=5:28.4.0-1~debian.13~trixie \

containerd.io=1.7.28-0~debian.13~trixie \

docker-buildx-plugin=0.28.0-0~debian.13~trixie \

docker-compose-plugin=2.39.4-0~debian.13~trixie

1. Uruchom Docker oraz skonfiguruj jego automatyczne uruchamianie wraz ze startem systemu operacyjnego. Wykorzystaj w tym celu poniższe polecenia.

sudo systemctl start docker

sudo systemctl enable docker

1. Dodaj użytkownika rbd do grupy docker wykonując polecenie:

sudo usermod -aG docker rbd

1. Wyloguj się i zaloguj ponownie w celu zaaplikowania zmian wprowadzonych w poprzednim punkcie. Uruchom ponownie terminal pomocniczy.
2. Zainstaluj k3d wykonując w terminalu pomocniczym poniższe polecenie:

wget -q -O - https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | TAG=v5.8.3 bash

1. Skonfiguruj menedżera pakietów do zainstalowania pakietu Kubernetes, wykonaj w tym celu poniższe polecenie:

sudo apt-get install -y apt-transport-https ca-certificates curl gnupg

curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.34/deb/Release.key \

| sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

sudo chmod 644 /etc/apt/keyrings/kubernetes-apt-keyring.gpg

echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.34/deb/ /' \

| sudo tee /etc/apt/sources.list.d/kubernetes.list

sudo chmod 644 /etc/apt/sources.list.d/kubernetes.list

1. Zainstaluj narzędzie kubectl, które umożliwi zarządzanie Kubernetes działającym w *k3d*. W tym celu uruchom poniższe polecenie:

sudo apt-get -y install kubectl=1.32.3+ds-2

1. W kolejnym kroku utwórz klaster węzłów *k3d*, wykorzystaj poniższe polecenie:

k3d cluster create RBDcluster --servers 1 --agents 4 --image rancher/k3s:v1.22.2-k3s1 \

-p "5432-5435:5432-5435@loadbalancer"

RBDcluster jest nazwą klastra. Klaster Kubernetes składa się z dwóch rodzajów węzłów: *control plane* (*server*  w nazewnictwie *k3d*), *worker node* (*agent* w nazewnictwie *k3d*). Agenty posiadają infrastrukturę przeznaczoną do uruchamiania kontenerów użytkownika. Zwiększanie liczby agentów głównie służy do zwiększanie skalowalności klastra przez rozpraszanie ich na wiele maszyn. Serwery zlecają zadania agentom, wykrywają i reagują na zdarzenia w klastrze (np. awarię agenta) oraz przechowują metadane opisujące stan klastra. Zwiększanie liczby serwerów służy głównie do zwiększania niezawodności działania funkcji systemowych klastra. Dodatkowym elementem klastra w dystrybucji *k3d* jest węzeł równoważenia obciążenia (*load balancer*), który zazwyczaj jest elementem infrastruktury dostawcy usług chmurowych. Węzeł ten odpowiedzialny za równomierne rozpraszanie żądań połączeń do replik usług użytkownika. Przełącznik *servers* służy do określenia liczby serwerów, przełącznik *agents* określa liczbę agentów, przełącznik *image* wskazuje na obraz kontenera zawierającego komponent *k3s*, przełącznik *port* umożliwia przekazywanie portów (ang. port forwarding) z węzłów klastra na adresy maszyny, na której jest uruchomiony *k3d*. W naszym przypadku przekazywane są 4 porty, od 5432 do 5433 z węzła loadbalancer. Zabieg ten umożliwia wykorzystanie adresu *localhost* do komunikacji z loadbalancer. Chociaż adres loadbalancer jest dostępny na maszynie gospodarza, to jego wartość nie jest z góry znana i jest dynamicznie ustalana podczas tworzenia klastra. Na potrzeby ćwiczeń łatwiej będzie posługiwać się adresem *localhost* maszyny gospodarza.

1. Uruchom poniższe polecenie aby sprawdzić jakie klastry zostały uruchomione w *k3d*:

k3d cluster list

1. Wyświetl listę węzłów klastrów *k3d*, uruchom poniższe polecenie;

k3d node list

1. Teraz przystąpimy do uruchomienia w klastrze pierwszej bazy danych Postgres. W pierwszym kroku pobierz plik manifestów za pomocą poniższego polecenia:

wget www.cs.put.poznan.pl/jjezierski/RBDv3/rbd1.yaml

1. Otwórz plik manifestów w celu jego przeglądnięcia za pomocą polecenia:

less rbd1.yaml

Plik manifestów jest opisany za pomocą języka [YAML](https://pl.wikipedia.org/wiki/YAML). Zwiera on deklarowany opis komponentów aplikacji wdrażanej w Kubernetes. W uproszeniu, język YAML jest hierarchicznym zestawem par klucz-wartość zwanych węzłami. Wartością może być skalar (tekst, liczba, wartość logiczna), uporządkowana sekwencja węzłów lub nieuporządkowany zbiór węzłów zwany mapą. Hierarchia węzłów jest wyznaczana przez wcięcia wierszy, analogicznie jak w języku Python. Elementy sekwencji zaznacza się znakiem myślnika. Plik rbd1.yaml zawiera 2 manifesty oddzielone trzema myślnikami i rozpoczynające się od klucza *apiVersion*, który określa wersję API Kubernetes wg której zastał zapisany manifest.

Podstawową jednostką wdrożenia i skalowania składnika aplikacji w Kubernetes jest *Pod*. Pod zawiera najczęściej jeden kontener. W pojedynczym Pod można umieścić wiele kontenerów jeżeli współdzielą ten sam zasób (np.: dysk), w takich sytuacjach często kontener udostępniający główną usługę jest wspomagany przez inne kontenery, które pełnią rolę pomocniczą, np.: stanowią adapter z innymi usługami lub inicjują główny kontener. Istnieje możliwość bezpośredniego zainstalowania Pod w klastrze, jednakże tak zainstalowany Pod nie jest kontrolowany przez Control Plane i jest pozbawiony funkcji automatycznego odtwarzania, skalowania oraz wersjonowanego aktualizowania i wycofywania zmian. W celu wyposażenia Pod w wyżej wymienione własności należy wdrożyć Pod za pomocą kontrolera. Kubernetes oferuje 3 rodzaje kontrolerów: *Deployment*, *StatefulSet* oraz *DeamonSet.* Pod wdrożony za pomocą *Deployment* jest całkowicie bezstanowy. W wyniku jego aktualizacji, skalowania, odtwarzania lub restartu Kubernetes może go fizycznie usunąć i utworzyć jego nową instancję. Jest to podejście bardzo elastyczne zapewniające dużą skalowalność i niezawodność, jednakże nie nadaje się dla składników aplikacji, które posiadają własne dane zwłaszcza dla baz danych. Do składowania danych istnieje możliwość zamontowania trwałego woluminu (PersistentVolume) w Pod przez kontroler Deployment, jednakże wszystkie repliki Poda współdzieliłyby ten wolumin co nie jest dozwolone dla baz danych typu *share nothing*, którego przedstawicielem jest Postgres. Kontroler *DeamonSet* uruchamia po jednej replice Pod na każdym węźle klastra. Ten kontroler najczęściej jest wykorzystywany dla usług kolekcjonujących dzienniki oraz monitorujących inne Pod. Wszystkie Pod kontrolowane za pomocą DeamonSet również współdzielą ten sam trwały wolumin i z tego powody ten rodzaj kontrolera nie jest przydatny do naszych celów. Kontroler *StatefulSet* jest przeznaczony do sterowania stanowymi Pod i z tego względu wykorzystuje się go uruchamiania baz danych i z tego powodu wykorzystamy go w dalszej części tutorialu.

Wracając do pliku manifestów. Pierwszy manifest opisuje Pod kontrolowany za pomocą StatefulSet zawiera jeden kontener z obrazem systemu Postgres. Klucz *kind* wskazuje na rodzaj użytego kontrolera, w tym przypadku StatefulSet. Klucz *metadata.name* został użyty do nazwania tego kontrolera jako pgsql-rbd1, nazwa ta musi być poprawną nazwą DNS ponieważ wchodzi w skład nazw replik Pod, które są rejestrowane w wewnętrznej usłudze klastra Kubernetes. Wszystkie węzły występujące bezpośrednio pod kluczem *spec* odnoszą się do specyfikacji kontrolera. Wartość klucza *spec.template* jest szablonem replik Pod, który jest wykorzystywany przez kontroler do tworzenia tych replik. W naszym przypadku szablon definiuje Pod składający się z jednego kontenera o nazwie *pgsql-rbd1*, który ma być utworzony z wykorzystaniem obrazu z repozytorium Docker o nazwie postgres:17.6. Każdy kontener ma zamontować trwały wolumin, którego nazwa rozpoczyna się od frazy *pgsql-rbd1-disk* w katalogu */data*. Definicje szablonów woluminów opisane są za pomocą klucza volumeClaimTemplates, który umożliwia utworzenie woluminu dla każdej repliki Pod. Klucz volumeClaimTemplates.spec określa między innymi tryb dostępu do woluminu oraz jego rozmiar. Klucz *env* umożliwia przekazanie do kontenera wartości zmiennych środowiskowych, które są zazwyczaj wykorzystywane przez kontener do swojej inicjalizacji. W naszym przypadku przekazano wartości zmiennych, które określają hasło użytkownika *postgres* oraz lokalizację katalogu zawierającego konfigurację i dane bazy danych. Zauważ, że katalog ten znajduje się na zamontowanym trwałym woluminie. Klucz *template.spec.node.selector* umożliwia określenie kryterium wyboru węzłów klastra, na których mają zostać uruchomione repliki *Pod*. W naszym przypadku posłużymy się nazwą hosta, aby uruchomić Pod na węźle k3d-rbdcluster-agent-0. Przypisanie Pod do węzłów klastra jest opcjonalne, rozpraszanie replik Pod w celu równoważenia obciążenia jest automatyczne.

Klucz spec.replicas wskazuje na liczbę żądanych replik Pod. W naszym przypadku wykorzystamy jedną replikę. Klucz spec.selector jest listą etykiet, którą muszą posiadać Pod aby kontroler mógł nimi zarządzać. Zauważ, że wartość klucza *spec.selector.matchLabels* (app: pgsql-rbd1) pasuje do wartości klucza *spec.template.metadata.labels*.

Repliki Pod, które zastaną utworzone przez kontroler mają dynamiczne adresy IP, które mogą się zmieniać przy skalowaniu lub odtwarzaniu Pod, dodatkowo adresy te działają w wewnętrznej sieci klastra. Z tego powodu trzeba utworzyć dodatkową usługę, która będzie miała stałe zewnętrzne IP za pośrednictwem której będzie można komunikować się z replikami Pod. W tym celu można wykorzystać *LoadBalancer*. Zadaniem tej usługi jest przekierowywanie połączeń do replik Pod w taki sposób aby równoważyć ich obciążenie.

Drugi manifest w pliku manifestów opisuje usługę *LoadBalancer*. Wartość klucza *kind* wskazuje, że manifest dotyczy usługi. Klucz *metadata.name* umożliwia nazwanie usługi. Węzeł *spec* określa specyfikację usługi. Klucz *spec.type* umożliwia wskazanie, że nasza usługa ma być typu *LoadBalancer*. Klucz *spec.ports* specyfikuje odwzorowanie portów, w naszym przypadku port 5432, na którym nasłuchuje baza danych Postgres uruchomiona w replikach Pod ma być odwzorowana w ten sam port adresów usługi *LoadBalancer*. Klucz *spec.selector* wskazuje etykiety Pod, dla których usługa *LoadBalancer* będzie wykonywać swoje zadanie.

Opuść program *less* wybierając przycisk *q*.

1. Standardowy obraz systemu Postgres nie zawiera jego rozszerzenia o nazwie pg\_fdw\_plus, które będzie potrzebne w dalszej części zajęć. Wydawałoby się, że można je zainstalować w działającym już kontenerze. Jednakże zmiany wykonane w plikach kontenera poza katalogami, w których są zamontowane trwałe woluminy są ulotne. Restart, skalowanie lub odtwarzanie Pod powoduje utratę zmian, które zostały wprowadzone do systemu plikowego kontenera przez instalator. Z tego powodu w pliku manifestu znajduje się referencja do obrazu nazwie rbd/postgres17:1.0, który przygotujemy w taki sposób, że będzie zawierał potrzebne nam rozszerzenia.
2. Pobierz plik *dockerfile* opisujący sposób konstrukcji obrazu rbd/postgres17:1.0 za pomocą poniższego polecenia:

wget http://www.cs.put.poznan.pl/jjezierski/RBDv3/pg\_fdw\_plus.dockerfile

1. Przeglądnij ten plik wykorzystując poniższe polecenie:

less pg\_fdw\_plus.dockerfile

Pierwsza dyrektywa FROM wskazuje obraz roboczy, w którym zostanie wykonana budowa binariów rozszerzenia pg\_fdw\_plus. Pierwsza dyrektywa RUN uruchamia sekwencję poleceń, które przygotowuje środowisko obrazu roboczego do budowania rozszerzenia. Druga dyrektywa RUN pobiera źródła rozszerzenia i buduje binaria rozszerzenia. Druga dyrektywa FROM wskazuje na źródłowy obraz, który wykorzystamy jako bazę dla naszego obrazu. Kolejne trzy dyrektywy COPY kopiują binaria rozszerzenia z obrazu roboczego do obrazu docelowego. Można byłoby zrezygnować z obrazu roboczego i wykonać budowanie rozszerzenia bezpośrednio w obrazie docelowym. Jednakże w takim przypadku obraz docelowy zawierałby środowisko deweloperskie co niepotrzebnie zwiększałoby rozmiar obraz.

Opuść program *less* wybierając przycisk *q*.

1. Rozpocznij wdrożenie komponentów z pliku manifestów za pomocą polecenia:

kubectl apply -f rbd1.yaml

1. Obserwuj postęp wdrożenia *StatefulSet* wykorzystując poniższe polecenie:

kubectl get sts --watch

Wdrożenie wymaga pobranie obrazu kontenera z repozytorium Docker w związku z tym zajmuje chwilę. Wdrożenie zakończy w momencie pojawienia się na terminalu wiersza, w którym liczba działających replik będzie równa liczbie żądanych replik, np.:  
pgsql-rbd1 1/1 1m

W tym momencie przerwij wykonanie polecenia wykorzystując kombinację Ctrl-c.

1. Teraz przygotujemy drugą bazę danych. Skopiuj plik manifestów do pliku rbd2.yaml:

cp rbd1.yaml rbd2.yaml

1. Użyj swojego ulubionego edytora tekstu wykonać następujące zmiany w pliku rbd2.yaml [Raport]:

* zamień wszystkie wystąpienia tekstu rbd1 na rbd2,
* zamień nazwę hosta k3d-rbdcluster-agent-0 na k3d-rbdcluster-agent-1,
* ustal w drugim manifeście wartość klucza spec.ports.port na 5433.

1. Wykonaj wdrożenie zawartości pliku rbd2.yaml [Raport].
2. Zainstaluj klienta Postgres wykonując następujące polecenia:

sudo apt-get -y install postgresql-client=17+278

1. Otwórz dwa kolejne okna lub zakładki terminala, w pierwszym ustaw znak zachęty za pomocą polecenia export PS1='[\u@rbd1 \W]\$ '. W drugim terminalu ustaw znak zachęty za pomocą polecenia export PS1='[\u@rbd2 \W]\$ '. Pierwszy terminal o nazwie *rbd1* będzie służył do operowania na bazie danych *RBD1*, natomiast drugi terminal o nazwie *rbd2* będzie wykorzystywany do wykonywania operacji na bazie danych *RBD2*.

## Transparentny dostęp do zdalnych danych

Celem tego punktu jest zaprezentowanie mechanizmów zdalnego serwera, odwzorowania użytkownika oraz importowania schematu, które umożliwiają transparenty dostęp do rozproszonych danych. Mechanizmy te są implementacją standardu ISO/IEC 9075-9 (SQL/MED).

1. W terminalu *rbd1* uruchom narzędzie *psql* logując się jako użytkownik *postgres* do bazy danych *RBD1*. Wykorzystaj polecenie

psql -U postgres -h localhost -p 5432 postgres

Użyj hasła rbd1 w celu uwierzytelnienia użytkownika postgres.

**Uwaga**: jeżeli narzędzie psql nieoczekiwanie traci połączenie ze serwerem zamiast adresu localhost użyj jednego z adresów węzła loadbalancer. W celu pozyskania tych adresów użyj polecenia: kubectl get svc. Wykorzystaj jeden z adresów z

kolumny EXTERNAL-IP.

1. W narzędziu *psql* ustaw znak zachęty dla pierwszego wiersza komendy za pomocą polecenia \set PROMPT1 '%n@RBD1:%>%# '.
2. W terminalu *rbd2* uruchom narzędzie *psql* logując się do bazy danych *RBD2*. Zwróć uwagę na podanie odpowiedniego numeru portu. W narzędziu *psql* ustaw odpowiedni znak zachęty.
3. W terminalu *rbd1* za pomocą narzędzia *psql* zainstaluj w bazie danych *RBD1* rozszerzenie [postgres\_fdw\_plus](https://github.com/pgfdwplus/postgres_fdw_plus), które jest forkiem rozszerzenia [postgres\_fdw](https://www.postgresql.org/docs/current/static/postgres-fdw.html) wykonując polecenie CREATE EXTENSION postgres\_fdw\_plus;.
4. W terminalu *rbd2* w bazie danych *RBD2* również zainstaluj rozszerzenie postgres\_fdw\_plus.
5. Pobierz skrypty SQL wykonując w terminalu pomocniczym polecenia:

wget www.cs.put.poznan.pl/jjezierski/RBDv3/pracownicy.sql

wget www.cs.put.poznan.pl/jjezierski/RBDv3/zespoly.sql

1. W bazie danych *RBD1* utwórz tabelę *pracownicy* uruchamiając skrypt w narzędziu *psql* poleceniem \i ~/pracownicy.sql.
2. W bazie danych *RBD2* utwórz tabelę *zespoly* uruchamiając skrypt ~/zespoly.sql.
3. Utwórz w bazie danych *RBD1* definicję [zdalnego serwera](https://www.postgresql.org/docs/current/static/sql-createserver.html) *rbd2* za pomocą poniższego polecenia:

CREATE SERVER rbd2

FOREIGN DATA WRAPPER postgres\_fdw

OPTIONS (host 'pgsql-rbd2-lb', port '5433', dbname 'postgres');

1. Utwórz w bazie danych *RBD1* [odwzorowanie zdalnego użytkownika](https://www.postgresql.org/docs/current/static/sql-createusermapping.html) *postgres* serwera *rbd2*, wykorzystaj poniższe polecenie:  
   CREATE USER MAPPING FOR postgres SERVER rbd2

OPTIONS (user 'postgres', password 'rbd2');

1. [Zaimportuj](https://www.postgresql.org/docs/current/static/sql-importforeignschema.html) do bazy danych *RBD1* fragment schematu *public* zawierający tabelę *zespoly* udostępniany przez serwer *rbd2*, wykorzystaj poniższe polecenie:

IMPORT FOREIGN SCHEMA public LIMIT TO (zespoly)

FROM SERVER rbd2 INTO public;

1. W terminalu *rbd1* wykonaj poniższe polecenia testujące zdalny dostęp do tabeli *zespoly*:

SELECT \* FROM zespoly;

SELECT nazwisko, nazwa FROM pracownicy natural join zespoly;

1. Utwórz w bazie danych RBD2 odpowiednie obiekty umożliwiające zdalny dostęp do tabeli pracownicy znajdującej się w bazie danych RBD1. [Raport]

## Rozproszone transakcje

Celem zadania jest przedstawienie rozproszonych transakcji, czyli takich transakcji, które modyfikują dane w więcej niż w jednej bazie danych. Istotnym problem do rozwiązania przez producentów rozproszonych systemów baz danych jest zapewnienie atomowości zatwierdzenia zmian wprowadzonych do różnych węzłów rozproszonej bazy danych. Problem ten rozwiązuje się wykorzystując wielofazowe algorytmy zatwierdzenia transakcji. Najpopularniejszym algorytmem jest algorytm dwufazowy o nazwie Two Phase Commit (2PC). Zapoznaj się z [tym artykułem](https://en.wikipedia.org/wiki/Two-phase_commit_protocol) opisującym protokół 2PC. Rozszerzenie postgres\_fdw\_plus wspiera ten protokół po odpowiedniej konfiguracji systemu.

### Przygotowanie systemu do realizacji rozproszonych transakcji

1. W terminalu *rbd1* w narzędziu *psql* zmień w pliku konfiguracyjnym bazy danych rbd1 wartość parametru max\_prepared\_transactions na wartość 10. Parametr ten określa liczbę lokalnych transakcji, które mogą uczestniczyć w globalnej (rozproszonej) transakcji. Wykorzystaj w tym celu poniższe polecenie:

ALTER SYSTEM set max\_prepared\_transactions =10;

1. W terminalu *rbd1* w narzędziu *psql* zmień w pliku konfiguracyjnym bazy danych rbd1 wartość parametru postgres\_fdw.two\_phase\_commit na wartość on. Parametr ten włącza obsługę protokołu 2PC dla globalnych transakcji realizowanych na zdalnych obiektach udostępnianych przez rozszerzenie pg\_fwd\_plus na poziomie wszystkich sesji system. Działanie tego parametru można ograniczyć do bieżącej sesji użytkownika, jeżeli ustawi się go za pomocą polecenia SET. Wykorzystaj w tym celu poniższe polecenie:

ALTER SYSTEM set postgres\_fdw.two\_phase\_commit = on;

1. W celach diagnostycznych włącz rejestrowanie wszystkich realizowanych przez Postgres poleceń SQL w pliku dziennika. W tym celu w terminalu *rbd1* w narzędziu *psql* wykonaj poniższe polecenia:

ALTER SYSTEM set log\_statement = 'all';

ALTER SYSTEM set log\_min\_duration\_statement = 0;

1. Zastosuj wprowadzone zmiany w pliku konfiguracyjnym bazy danych *rbd1* przez wykonanie w terminalu *rbd1* następującego polecenia:

select pg\_ctl reload();

1. Powtórz powyższe polecenia w bazie *rbd2*.

### Rozproszona transakcja zakończona sukcesem

1. Otwórz kolejną zakładkę terminala i nazwij ją *diag-rbd1*. Uruchom w niej poniższe polecenie aby wyświetlać informacje diagnostyczne bazy danych *rdb1*:

kubectl logs -f pgsql-rbd1-0

1. Powtórz poprzedni krok dla bazy danych *rbd2*.
2. W terminalu *rbd1* za pomocą narzędzia *psql* wykonaj w bazie danych RBD1 poniższą transakcję rozproszoną:

begin;

update zespoly set adres='PIOTROWO 1' where id\_zesp=10;

update pracownicy set placa\_pod=999 where id\_prac=100;

commit;

1. Sprawdź w terminalu *rbd2* stan tabel *pracownicy* i *zespoly*. Czy posiadają one zmiany wprowadzone przez transakcję z poprzedniego punkty? [Raport]
2. Sprawdź zawartość terminala *diag-rbd1*. W informacjach diagnostycznych powinny być wyświetlone informacje o wszystkich czterech poleceniach zleconych w ramach transakcji z poprzedniego punktu.
3. Sprawdź zawartość terminala *diag-rbd2*. W informacjach diagnostycznych powinny być wyświetlone między innymi poniższe informacje:  
   START TRANSACTION ISOLATION LEVEL READ COMMITTED

parse <unnamed>: UPDATE public.zespoly ...

bind <unnamed>: UPDATE public.zespoly ...

execute <unnamed>: UPDATE public.zespoly ...

PREPARE TRANSACTION 'pgfdw\_803\_16437\_84\_null'

COMMIT PREPARED 'pgfdw\_803\_16437\_84\_null'

Powyższe informacje są śladem wykonania poleceń lokalnej transakcji realizowanej w bazie danych *rdb2*. Pierwsze polecenie rozpoczyna lokalną transakcję w trybie izolacji *read committed*. Kolejne 3 operacje reprezentują trzy fazy wykonania polecenia UPDATE na tabeli *zespoly*. Kolejne 2 polecenia są kluczowe z punktu widzenia realizacji protokołu 2PC. Polecenie PREPARE TRANSACTION przygotowuje lokalną transakcję bazy danych *rbd2* do zatwierdzenia. Polecenie COMMIT PREPARED zatwierdza wcześniej przygotowaną transakcję.

Zauważ, globalna rozproszona transakcja uruchomiona w bazie danych *rbd1* nie spowodowała wykonania poleceń PREPARE TRANSACTION i COMMIT PREPARED w bazie danych *rbd1*. Z tego wynika, że baza danych *rbd1* nie weszła w stan PREPARED. Jest to typowe dla wariantu algorytmu 2PC, w którym jedna z baz danych nazywana węzłem zatwierdzania zachowuje się w ten sposób.

W powyższym scenariuszu transakcji baza danych *rbd1* pełni rolę zarówno koordynatora jak i węzła zatwierdzania. Po odebraniu polecenia COMMIT zatwierdzającego globalną transakcję baza danych *rbd1* wysyła do bazy danych *rbd2* polecenie PREPARE TRANSACTION. Jeżeli *rbd1* odbierze od *rbd2* potwierdzenie wykonania operacji, to zatwierdza swoją transakcję i następnie wysyła do *rbd2* polecenie COMMIT PREPARED. W ten sposób globalna rozproszona transakcja kończy się sukcesem.

### Rozproszona transakcja zakończona awarią przed fazą PREPARE

Teraz zakłócimy pomyślną ścieżkę protokołu 2PC przez restart bazy danych *rdb2* przed wykonaniem polecenia COMMIT w bazie *rbd1* i sprawdzimy zachowanie obu baz danych.

1. W terminalu *rbd1* za pomocą narzędzia *psql* wykonaj w bazie danych RBD1 rozpocznij transakcję rozproszoną:

begin;

update zespoly set adres='PIOTROWO 3' where id\_zesp=10;

update pracownicy set placa\_pod=888 where id\_prac=100;

1. Otwórz nowe okno terminala, które dalej będzie nazywane pomocniczym. Zrestartuj Pod pgsql-rbd2, w tym celu wykonaj w terminalu pomocniczym poniższe polecenie:

kubectl rollout restart sts pgsql-rbd2

1. Spróbuj zatwierdzić globalną transakcję wydając w terminalu *rbd1* polecenie:  
   commit;

Uzyskasz informację o błędzie:

ERROR: FATAL: terminating connection due to administrator command

W terminalu diag-rbd1 znajdziesz rozszerzoną informację:

ERROR: FATAL: terminating connection due to administrator command

CONTEXT: remote SQL command: PREPARE TRANSACTION ...

Błąd pojawił się przy próbie wykonania zdalnej operacji PREPARE TRANSACTION na bazie danych *rbd2*.

1. W terminalu *rbd1*  sprawdź identyfikator bieżącej transakcji wykonując polecenie:

SELECT txid\_current\_if\_assigned();

Zapewne dostałeś wartość pustą co oznacza, że twoja sesja nie ma aktywnej transakcji.

W takim razie co się stało z twoją transakcją? Trwa nadal, została zatwierdzona czy wycofana?

Sprawdź stan modyfikowanego wiersza tabeli *pracownicy* wydając polecenie:

select placa\_pod from pracownicy where id\_prac=100 for update nowait;

Okazuje się, że płaca nie została zmieniona.

Polecenie to nie tylko odczytuje dane ale użycie klauzuli FOR UPDATE NOWAIT również blokuje odczytywany wiersz w trybie wyłącznym. W związku z tym, że polecenie to zostało wydane poza zakresem transakcji to założona blokada została po wykonaniu polecenia natychmiast zwolniona. Jeżeli jakaś transakcja blokowałaby odczytywany wiersz, to polecenie SELECT zakończyłoby się błędem. W związku z tym, że odczyt został wykonany poprawnie i płaca nie została zmieniona może wnioskować, że lokalna transakcja w bazie danych *rbd1* została wycofana.

Teraz sprawdzimy co się stało z lokalną transakcją w bazie danych *rbd2.*

1. W terminalu diag-rbd2 zrestartuj poniższe polecenie, które zostało przerwane z powodu restartu Poda:

kubectl logs -f pgsql-rbd1-0

W terminalu zobaczysz podobne wpisy do poniższych:

LOG: received fast shutdown request

LOG: aborting any active transactions

Wpisy wskazują, że wszystkie aktywne transakcje, w tym lokalna transakcja na bazie *rdb2* wchodząca w skład testowanej globalnej transakcji, zostały wycofane.

1. Wykonamy jeszcze jedno sprawdzenie, podobne jak w przypadku bazy danych *rbd1*. W terminalu *rbd2* w narzędziu *psql* nawiąż ponownie połączenie do bazy danych *RBD2* za pomocą polecenia: \c postgres postgres localhost 5433.
2. W terminalu *rbd2* wykonaj poniższe polecenie:

select adres from zespoly where id\_zesp=10 for update nowait;

Polecenie wykonało się poprawnie i adres zespołu nie zmienił się, co potwierdza, że lokalna transakcja na bazie danych *rbd2* została wycofana.

Ustaliliśmy, że zmiany w bazie danych *rbd1* i *rbd2* zostały wycofane co oznacza, że cała globalna transakcja została wycofana. W ten sposób potwierdziliśmy, że implementacja protokołu 2PC wykonana w rozszerzeniu pg\_fwd\_plus działa poprawnie w przetestowanym scenariuszu awarii.

1. Przetestuj analogiczny scenariusz awarii, w którym zasymulujesz awarię bazy danych *rbd1* przed fazą PREPARE. [Raport]

### Rozproszona transakcja zakończona awarią po fazie PREPARE

Teraz zakłócimy pomyślną ścieżkę protokołu 2PC po wykonaniu polecenia PREPARE TRANSACTION w bazie *rbd2* i sprawdzimy zachowanie obu baz danych.

1. Do zakłócenia zatwierdzenia globalnej transakcji wykorzystamy parametr *skip\_commit\_phase* rozszerzenia *postgres\_fdw\_plus*. Ustawienie tego parametru na wartość *true* spowoduje, że przy wykonaniu polecenia COMMIT globalnej transakcji do zdalnych baz danych jest wysyłane PREPARE TRANSACTION, następnie lokalna transakcja w węźle zatwierdzania jest zatwierdzana ale do zdalnych baz danych nie jest wysyłane polecenie COMMIT PREPARED. Parametr ten umożliwia zasymulowanie awarii zdalnych baz danych po wykonaniu polecenia PREPARE TRANSACTION ale przed wykonaniem COMMIT PREPARED. W terminalu *rbd1* za pomocą narzędzia *psql* wykonaj następujące polecenie:

set postgres\_fdw.skip\_commit\_phase=true;

1. W terminalu *rbd1* za pomocą narzędzia *psql* wykonaj w bazie danych RBD1 wykonaj transakcję rozproszoną:

begin;

update zespoly set adres='PIOTROWO 3' where id\_zesp=10;

update pracownicy set placa\_pod=888 where id\_prac=100;

commit;

1. W terminalu *rbd1* sprawdź stan modyfikowanego wiersza tabeli *pracownicy* wydając polecenie:

select placa\_pod from pracownicy where id\_prac=100 for update nowait;

Okazuje się, że zmiany płacy pracownika zostały wprowadzone do bazy danych *rbd1* i zatwierdzone. Z tego wynika, że lokalna transakcja realizowana w tej bazie danych została zatwierdzona.

Teraz sprawdzimy co się stało z lokalną transakcją w bazie danych *rbd2.*

1. Przeglądnij wpisy w terminalu diag-rbd2. Wpisy wskazują, że zostało wykonane polecenie PREPARE TRANSACTION ale nie zostało wykonane polecenie COMMIT PREPARED.
2. Wykonamy jeszcze jedno sprawdzenie, podobne jak w przypadku bazy danych *rbd1*. W terminalu *rbd2* w narzędziu *psql* nawiąż ponownie połączenie do bazy danych *RBD2* za pomocą polecenia: \c postgres postgres localhost 5433.
3. Sprawdź jaki status ma modyfikowany wiersz tabeli *zespoly*, w tym celu w terminalu *rbd2* wykonaj poniższe polecenie:

select adres from zespoly where id\_zesp=10 for update nowait;

Polecenie zakończyło się błędem, co oznacza, że lokalna transakcja nie została zakończona.

1. W terminalu *rbd2* wykonaj poniższe polecenie aby wyświetlić informacje o transakcjach bazy danych *rbd2*, które znajdują się w stanie PREPARE:

select \* from pg\_prepared\_xacts;

Z resultatu powyższego zapytania wynika, że lokalna transakcja bazy danych *rdb2* znajduje się w stanie PREPARED.

1. W celu zwiększenia realizmu scenariusza awarii zasymuluj awarię bazę danych przez restart jej Poda, w tym celu w terminalu pomocniczym wykonaj polecenie:

kubectl rollout restart sts pgsql-rbd2

1. W terminalu diag-rbd2 zrestartuj poniższe polecenie, które zostało przerwane z powodu restartu Poda:

kubectl logs -f pgsql-rbd1-0

W terminalu zobaczysz między innymi wpis:

LOG: recovering prepared transaction … from shared memory

Wpis ten oznacza, że lokalna transakcja bazy danych *rbd2* znajdująca się w stanie PRAPARED została odzyskana.

* 1. W terminalu *rbd2* w narzędziu *psql* nawiąż ponownie połączenie do bazy danych *RBD2* za pomocą polecenia: \c postgres postgres localhost 5433.
  2. W terminalu *rbd2* ponownie sprawdź status wiersza tabeli *zespoly* wykorzystując polecenie SELECT … FOR UPDATE NOWAIT. Sprawdź też zawartość tabeli *pg\_prepared\_xacts*. Okazuje się, że restart bazy danych symulujący jej awarię nic nie zmienił: modyfikowany wiersz jest zablokowany przez odtworzoną lokalną transakcję i ta transakcja jest cały czas w stanie PREAPARED.
  3. Wiemy, że lokalna transakcja w bazie danych *rbd1*, czyli w węźle zatwierdzenia została zatwierdzona. W celu zachowania atomowości globalnej transakcji, lokalna transakcja w bazie danych *rbd2* również powinna zostać zatwierdzona. Możemy to zrobić przez wykonanie polecenia COMMIT ROLLBACK *gid*, gdzie *gid* to identyfikator pozyskany z kolumny o tej samej nazwie z tabeli *pg\_prepared\_xacts*. Alternatywnie można skorzystać z funkcji *pgfdw\_plus\_resolve\_foreign\_prepared\_xacts* dostarczanej przez rozszerzenie *postgres\_fdw\_plus*. Funkcja ta korzysta z tabeli *pgfdw\_plus.xact\_commits*, w której koordynator (czyli w naszym przypadku baza danych *rbd1*) składuje identyfikatory lokalnych transakcji. Funkcja wymaga rozszerzenia *dblink*, w celu jego zainstalowania wykonaj w terminalu *rbd1* poniższe polecenie:

create extension dblink;

* 1. Wykorzystaj funkcję *pgfdw\_plus\_resolve\_foreign\_prepared\_xacts* do zakończenia globalnej transakcji, w tym celu wykonaj w terminalu *rbd1* poniższe polecenie:

SELECT \* FROM pgfdw\_plus\_resolve\_foreign\_prepared\_xacts('rbd2', false);

W terminalu *rbd2* ponownie sprawdź status wiersza tabeli *zespoly* wykorzystując polecenie SELECT … FOR UPDATE NOWAIT. Polecenie wykonało się poprawnie i adres zespołu zmienił się, co potwierdza, że lokalna transakcja w bazie danych *rbd2* została zatwierdzona. Sprawdź też zawartość tabeli *pg\_prepared\_xacts*. W bazie danych *rbd2* nie ma już żadnych transakcji w stanie PREPARED. Ustaliliśmy, że zmiany w bazie danych *rbd1* i *rbd2* zostały zatwierdzone co oznacza, że cała globalna transakcja została zatwierdzona. W ten sposób potwierdziliśmy, że implementacja protokołu 2PC wykonana w rozszerzeniu pg\_fwd\_plus działa poprawnie w przetestowanym scenariuszu awarii.

## Rozproszone zakleszczenie

System Postgresql, jak praktycznie wszystkie systemy zarządzania bazami danych, do synchronizacji transakcji wykorzystuje blokady transakcyjne. Celem ćwiczenia jest sprawdzenie w jaki sposób system ten radzi sobie z rozproszonym zakleszczeniem.

1. W bazie danych *RBD1* rozpocznij rozproszoną transakcję T1 za pomocą poniższych poleceń SQL:

begin;

update pracownicy set placa\_pod=placa\_pod+10 where id\_prac=100;

1. W bazie danych *RBD2* rozpocznij rozproszoną transakcję T2 za pomocą poniższych poleceń SQL:

begin;

update zespoly set adres='PIOTROWO 88' where id\_zesp=10;

1. W bazie danych *RBD1* kontynuuj rozproszoną transakcję T1 za pomocą poniższego polecenia SQL:

update zespoly set adres='PIOTROWO 77' where id\_zesp=10;

1. W bazie danych *RBD2* kontynuuj rozproszoną transakcję T2 za pomocą poniższego polecenia SQL:

update pracownicy set placa\_pod=placa\_pod+10 where id\_prac=100;

1. Co się stało? Poczekaj jeszcze 1 minutę. Coś się zmieniło?
2. Niestety system Postgresql nie wykrywa rozproszonych zakleszczeń.
3. W terminalu *rbd2* za pomocą klawiszy Ctr-C przerwij oczekiwanie na blokadę.
4. W terminalu *rbd1* dokończ transakcję T1 za pomocą polecenia SQL COMMIT;.
5. W terminalu *rbd2* spróbuj zatwierdzić transakcję T2. Co się stało?
6. W terminalu *rbd2* zrestartuj całą transakcję T2 przewidzianą dla bazy danych RBD2 i zatwierdź transakcję.
7. W celu uniknięcia zakleszczenia przed pierwszym poleceniem modyfikującym zdalny obiekt należałoby wykonać polecenie SET LOCAL statement\_timeout = 10000, które na czas do końca transakcji ogranicza czas wykonania dalszych poleceń w transakcji do wskazanego w milisekundach czasu (w tym przypadku 10s).
8. Zrealizuj powyższe transakcje wykorzystując parametr statement\_timeout w celu uniknięcia zakleszczenia. Jakie wady posiada to rozwiązanie? [Raport]

## Prosta replikacja danych

Replikacja jest procesem powielania tych samych danych w różnych bazach danych w rozproszonym systemie informatycznym. Proces ten zwiększa niezawodność dostępu do danych oraz wydajność. Celem zadania jest zapoznanie się z prostym mechanizmem asynchronicznej replikacji typu master-slaves implementowanej za pomocą [materializowanych perspektyw](https://www.postgresql.org/docs/current/static/sql-creatematerializedview.html). Asynchroniczność replikacji oznacza, że uaktualnianie zawartości materializowanej perspektywy następuje niezależnie od transakcji modyfikującej dane źródłowe. W replikacji master-slaves dane mogą być modyfikowane jedynie w bazie danych zawierającej tabele źródłowe, repliki służą jedynie do odczytu.

1. W bazie danych RBD1 utwórz materializowaną perspektywę za pomocą poniższego polecenia:

CREATE MATERIALIZED VIEW zespoly\_replika AS SELECT \* FROM zespoly;

1. Sprawdź zawartość materializowanej perspektywy *zespoly\_replika*.
2. W bazie danych RBD2 wykonaj poniższe polecenie w celu zmiany danych źródłowych w tabeli *zespoly*.

update zespoly set adres='PIOTROWO 43' where id\_zesp=10;

1. Sprawdź zawartość materializowanej perspektywy *zespoly\_replika*. Czy się zmieniała? Dlaczego?
2. W bazie danych RBD1 [odśwież zawartość materializowanej](https://www.postgresql.org/docs/current/static/sql-refreshmaterializedview.html) perspektywy za pomocą poniższego polecenia:

REFRESH MATERIALIZED VIEW zespoly\_replika;

1. Sprawdź zawartość materializowanej perspektywy *zespoly\_replika*. Jaka jest wada takiego sposobu odświeżania materializowanych perspektyw? [Raport]
2. W następnym etapie będziemy automatycznie, cyklicznie odświeżać materializowaną perspektywę. W tym celu zainstalujemy [rozszerzenie pg\_cron](https://github.com/citusdata/pg_cron/), które nie znajduje się w standardowej dystrybucji systemu Postgres.

Przygotujemy kolejną wersję naszego obrazu Docker systemu Postgres.

1. Pobierz plik *dockerfile* za pomocą poniższego polecenia:

wget http://www.cs.put.poznan.pl/jjezierski/RBDv3/pg\_cron.dockerfile

1. Przeglądnij ten plik wykorzystując poniższe polecenie:

less pg\_cron.dockerfile

Dyrektywa FROM wskazuje na źródłowy obraz, który wykorzystamy jako bazę dla naszego obrazu, skorzystamy z obrazu rbd/postgres17:1.0, który przygotowaliśmy wcześniej. Dyrektywa RUN uruchamia polecenie wewnątrz źródłowego obrazu. Wykorzystamy sekwencję poleceń, które zainstaluje pliki rozszerzenia pg\_cron w systemie plikowym naszej wersji obrazu. Zamiast wielu dyrektyw RUN do uruchomienia każdego polecenia instalacji oddzielnie zastosowano sekwencję poleceń w jednej dyrektywie. Takie rozwiązanie minimalizuje liczbę warstw obrazu kontenera, które są generowane przez każdą dyrektywę, co optymalizuje rozmiar docelowego obrazu. Zwróć uwagę na odmienny sposób instalacji rozszerzenia postgres\_fdw\_plus i pg\_cron. Wynika z faktu, że rozszerzenie pg\_cron jest w repozytorium Postgresa dostępne w postaci binarnej przeznaczonej na wersję Linux obrazu docelowego, natomiast rozszerzenie pg\_fdw\_plus jest dostępne jedynie w wersji źródłowej.

Opuść program *less* wybierając przycisk *q*.

1. Utwórz wersję 1.1 obrazu wykonując polecenie:

docker build -t rbd/postgres17:1.1 - < pg\_cron.dockerfile

Przełącznik *t* polecenia *docker build* służy do wskazania nazwy i wersji naszego obrazu.

1. Utworzony obraz jest dostępny jedynie lokalnie. W celu zaimportowania go do klastra wykonaj poniższe polecenie:

k3d image import rbd/postgres17:1.1 --cluster RBDcluster

1. Zmodyfikuj plik manifestów rbd1.yaml, ustaw wartość klucza *image* na wartość *rbd/postgres17:1.1* w celu zastosowania własnego obrazu kontenera.
2. Wykonaj wdrożenie z wykorzystaniem zmodyfikowanego pliku manifestów. W środowisku wykonawczym zostaną wprowadzone zmiany jedynie wynikające ze zmian w manifeście. W szczególności nie zostaną zmienione trwałe woluminy, dzięki temu stan bazy danych *rbd1* jest ten sam jak przed wdrożeniem zmian.
3. Zrestartuj połączenie z psql do bazy danych RBD1.
4. W celu załadowania do Postgresa zainstalowanej w obrazie biblioteki pg\_cron wykonaj w bazie RBD1 następujące polecenie:

alter system set shared\_preload\_libraries = 'pg\_cron';

1. W terminalu pomocniczym zrestartuj Pod pgsql-rbd1, w celu zaaplikowanie zmian wprowadzonych w poprzednim kroku.
2. Zrestartuj połączenie z psql do bazy danych RBD1.
3. Utwórz w bazie danych RBD1 rozszerzenie pg\_cron wykonując w psql poniższe polecenie:

CREATE EXTENSION pg\_cron;

1. Powtórz kroki od 12 do 18 dla bazy danych RBD2.
2. Wykorzystaj dokumentację [rozszerzenia pg\_cron](https://github.com/citusdata/pg_cron/) do przygotowanie polecenia, które automatycznie, cyklicznie (np. co 5 minut) będzie odświeżać materializowaną perspektywę zespoly\_replika [Raport].
3. Usuń klaster RBDcluster wykonując w terminalu pomocniczym polecenie:

k3d cluster delete RBDcluster